CSCI 5417
Information Retrieval Systems

Jim Martin

Lecture 6
9/8/2011

Today 9/8

- Review basic Vector Space Model
  - TF*IDF weighting
  - Cosine scoring
  - Ranked retrieval
- More efficient scoring/retrieval

Summary: tf x idf (or tf.idf)

- Assign a tf.idf weight to each term *i* in each document *d*

$$w_{t,d} = tf_{t,d} \times \log(N / df_t)$$

$tf_{t,d}$ = frequency of term t in document d

N = total number of documents

df_t = the number of documents that contain term t

- Weight increases with the number of occurrences *within* a doc
- And increases with the rarity of the term *across* the whole corpus

Real-valued Term Vectors

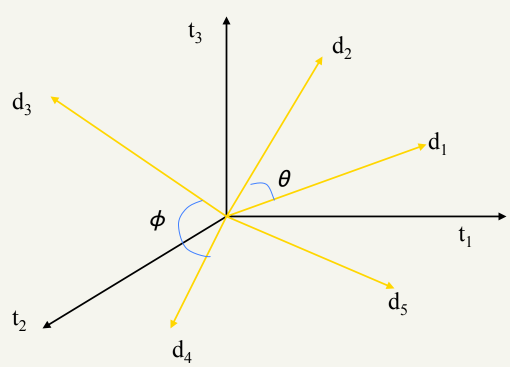
- Still Bag of words model
- Each is a vector in $\mathbb{R}^M$
  - Here log-scaled *tf.idf*

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	13.1	11.4	0.0	0.0	0.0	0.0
Brutus	3.0	8.3	0.0	1.0	0.0	0.0
Caesar	2.3	2.3	0.0	0.5	0.3	0.3
Calpurnia	0.0	11.2	0.0	0.0	0.0	0.0
Cleopatra	17.7	0.0	0.0	0.0	0.0	0.0
mercy	0.5	0.0	0.7	0.9	0.9	0.3
worser	1.2	0.0	0.6	0.6	0.6	0.0

Documents as Vectors

- Each doc *j* can now be viewed as a vector of *wf* values, one component for each term
- So we have a vector space
  - Terms are axes
  - Docs live in this space
  - Number of dimensions is the size of the dictionary
- And for later...
  - Terms (rows) are also vectors
  - Docs are the dimensions

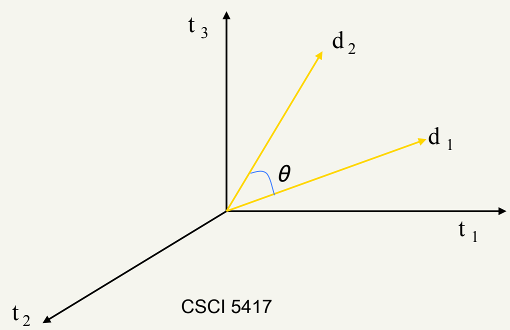
Intuition



Documents that are "close together" in the vector space talk about the same things.

Cosine Similarity

- Distance between vectors d_1 and d_2 *captured* by the cosine of the angle x between them.



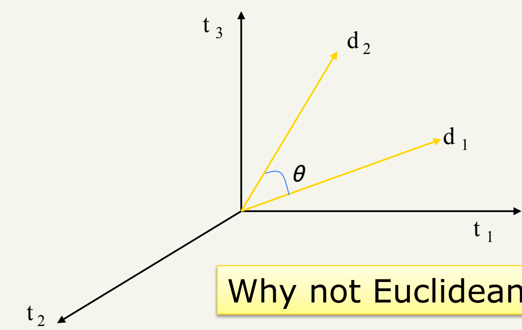
The Vector Space Model

Queries are just short documents

- Take the freetext query as short document
- Return the documents ranked by the closeness of their vectors to the query vector.

Cosine Similarity

Similarity between vectors d_1 and d_2 *captured* by the cosine of the angle x between them.



Why not Euclidean distance?

9/22/11

Cosine similarity

$$sim(d_j, d_k) = \frac{\vec{d}_j \cdot \vec{d}_k}{\left|\vec{d}_j\right| \left|\vec{d}_k\right|} = \frac{\sum_{i=1}^{M} w_{i,j} w_{i,k}}{\sqrt{\sum_{i=1}^{M} w_{i,j}^2} \sqrt{\sum_{i=1}^{M} w_{i,k}^2}}$$

- Cosine of angle between two vectors
  - The denominator involves the lengths of the vectors.

Normalization

Normalized vectors

- For normalized vectors, the cosine is simply the dot product:

$$\cos(\vec{d}_j, \vec{d}_k) = \vec{d}_j \cdot \vec{d}_k$$

So...

- Basic ranked retrieval scheme is to
  - Treat queries as vectors
  - Compute the dot-product of the query with *all* the docs
  - Return the ranked list of docs for that query.

But…

- What do we know about the document vectors?
- What do we know about query vectors?

Scoring

(1) N documents. Each gets a score.

(2) Get the lengths for later use

(3) Iterate over the query terms

(6) Accumulate the scores for each doc, a term at a time

(9) Normalize by doc vector length

```
CosineScore(q)
1  float Scores[N] = 0
2  Initialize Length[N]
3  for each query term t
4  do calculate w_{t,q} and fetch postings list for t
5     for each pair(d, tf_{t,d}) in postings list
6     do Scores[d] += wf_{t,d} × w_{t,q}
7  Read the array Length[d]
8  for each d
9  do Scores[d] = Scores[d]/Length[d]
10 return Top K components of Scores[]
```

Scoring

```
CosineScore(q)
float Scores[N] = 0
Initialize Length[N]
for each query term t
do calculate w_{t,q} and fetch postings list for t
    for each pair(d, tf_{t,d}) in postings list
    do Scores[d] += wf_{t,d} × w_{t,q}
Read the array Length[d]
for each d
do Scores[d] = Scores[d]/Length[d]
return Top K components of Scores[]
```


Note

- That approach is know as term at a time scoring... For obvious reasons
- An alternative is document at a time scoring where
  - First you do a Boolean AND or OR to derive a candidate set of docs
  - Then you loop over those docs scoring each in turn by looping over the query terms
- Pros and cons to each one.

Speeding that up

- Two basic approaches...
  - Optimize the basic approach by focusing on the fact that queries are short and simple
    - Make the cosines faster
    - Make getting the top K efficient
  - Give up on the notion of finding the best K results from the total N
    - That is, let's approximate the top K and not worry if we miss some docs that should be in the top K

More Efficient Scoring

- Computing a single cosine efficiently
- Choosing the *K* largest cosine values efficiently.
  - Can we do this without computing all *N* cosines?
  - Or doing a sort of N things

Efficient Cosine Ranking

- What we're doing in effect: solving the *K*-nearest neighbor problem for a query vector
- In general, we do not know how to do this efficiently for high-dimensional spaces
- But it is solvable for short queries, and standard indexes support this well

Special case – unweighted queries

- No weighting on query terms
  - Assume each query term occurs only once
    - TF is 1
    - Ignore IDF in the query weight

Faster cosine: unweighted query

```
FastCosineScore(q)
float Scores[N] = 0
for each d
do Initialize Length[d] to the length of doc d
for each query term t
do calculate w_{t,q} and fetch postings list for t
    for each pair(d, tf_{t,d}) in postings list
    do add wf_{t,d} to Scores[d]
Read the array Length[d]
for each d
do Divide Scores[d] by Length[d]
return Top K components of Scores[]
```

Figure 7.1 A faster algorithm for vector space scores.

Computing the *K* largest cosines: selection vs. sorting

- Typically we want to retrieve the top *K* docs (in the cosine ranking for the query)
  - not to totally order all docs in the collection
  - $K << N$
- Use a heap

Quiz

- Quiz 1 is September 27
- Chapters 1-4, 6-9, and 12 will be covered
  - As of today you should have read Chapters 1-4, 6 and 7.
  - I'll provide relevant page ranges
  - Material in the book not covered in class will be on the quiz.
- Old quizzes are posted
  - Try to work through them; mail me if you get stuck

Approximation

- Cosine (-ish) scoring is still to expensive. Can we avoid all this computation?
- Yes, but may sometimes get it wrong
  - a doc *not* in the top *K* may creep into the list of *K* output docs
    - And a doc that should be there isn't there
  - Not such a bad thing

Cosine Similarity is a Convenient Fiction

- User has a task and a query formulation
- Cosine matches docs to query
- Thus cosine is just a proxy for user happiness
- If we get a list of K docs "close" to the top K by cosine measure, we should be ok

Generic approach

- Find a set A of *contenders*, with
  - $K < |A| << N$
  - A does not necessarily contain the top K, but has many docs from among the top K
  - Return the top K docs from set A
- Think of A as eliminating likely non-contenders

Candidate Elimination

- Basic cosine algorithms consider all docs containing at least one query term
  - Because of the way we loop over the query terms
    - For each query term
      - For each doc in that terms postings
- To cut down on this we could short-circuit the outer loop or the inner loop or both

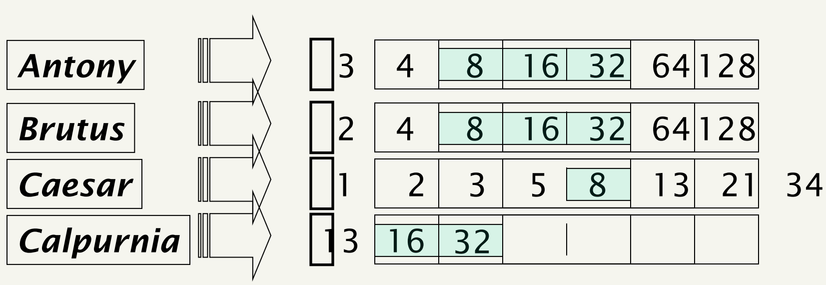
High-IDF Query Terms Only

- For a query such as *catcher in the rye*
- Only accumulate scores from *catcher* and *rye*
  - *in* and *the* contribute little to the scores and don't alter rank-ordering much
- Benefit:
  - Postings of low-idf terms have many docs → these (many) docs get eliminated from *A*

Docs containing many query terms

- For multi-term queries, only compute scores for docs containing several of the query terms
  - Say, at least 3 out of 4
  - Imposes a "soft conjunction" on queries seen on web search engines (early Google)
- Easy to implement in postings traversal

3 of 4 query terms



Scores only computed for docs8, 16 and 32.

Champion Lists

- Precompute for each dictionary term *t*, the *r* docs of highest weight in *t*'s postings
  - Call this the champion list for *t*
    - AKA fancy list or top docs for *t*
- At query time, only compute scores for docs in the champion list of some query term
  - Pick the *K* top-scoring docs from among these

Early Termination

- When processing query terms look at them in order of decreasing idf
  - High idf terms likely to contribute most to score
- As we update the score contribution from each query term, stop if doc scores are relatively unchanged

Next time

- Start on evaluation